NBSIR 83-2740

Specification and Interpretation of Data Model Semantics: An Integration of Two Approaches

US. DEPARTMENT OF COMMERCE
National Bureau of Standards
Center for Programming Science and Technology
Institute for Computer Sciences and Technology
Washington, DC 20234

July 1983

S. DEPARTMENT OF COMMERCE
ATIONAL BUREAU OF STANDARDS

NBSIR 83-2740

SPECIFICATION AND INTERPRETATION OF DATA MODEL SEMANTICS: AN INTEGRATION OF TWO APPROACHES

Winfried Lamersdorf

U.S. DEPARTMENT OF COMMERCE
National Bureau of Standards
Center for Programming Science and Technology
Institute for Computer Sciences and Technology
Washington, DC 20234

July 1983

U.S. DEPARTMENT OF COMMERCE, Malcolm Baldrige, *Secretary*
NATIONAL BUREAU OF STANDARDS, Ernest Ambler, *Director*

SPECIFICATION AND INTERPRETATION OF DATA MODEL SEMANTICS : AN INTEGRATION OF TWO APPROACHES

Winfried Lamersdorf

Universität Hamburg
Fachbereich Informatik
Schlüterstrasse 70
D-2000 Hamburg 13
West-Germany

November 1982

This work was done, in part, while in residence at the Data Management and Programming Languages Division, Center for Programming Science and Technology, Institute for Computer Sciences and Technology, National Bureau of Standards, Washington, D.C. 20234

TABLE OF CONTENTS

SPECIFICATION AND INTERPRETATION OF DATA MODEL SEMANTICS : AN INTEGRATION OF TWO APPROACHES

Winfried Lamersdorf

Two different approaches to database model semantics description and evaluation are presented, compared, and integrated: a formal semantic specification method as originally developed for programming languages, and a computer-based data model interpreter to be used as a rapid prototyping system. Both ways to specify database model semantics are applied to a common example.

Two alternatives to combine the advantages of both methods are analyzed in detail: First, it is shown how the semantics of the data model processor can be precisely described in terms of the formal semantic specification method. Then, it is demonstrated how the abstract meta-language of the specification method can be mapped to the executable commands of the data model processor. Thus, database semantics can both be specified in an abstract and high-level way and still be analyzed and evaluated automatically.

Key words: databases; data models; data model processing; data model prototyping; data model semantics; denotational semantics; formal semantic specification; relational database; relational data model; semantic model interpreter.

1. INTRODUCTION

Designing the software tools for major database applications has become an increasingly complex task. This affects not only the development of special purpose database application programs, but also the design and specification of appropriate query languages as well as adequate, new data models.

To assist in the process of designing, specifying, and evaluating the semantics of data models and software components, new tools have been developed recently. Two major directions can be separated: first, formal methods for the precise specification of programming language semantics have been successfully applied to the design and specification of, e.g., a database query language [7], and a major data model [9]. On the other hand, methods derived from recent developments of 'rapid prototyping' systems can be applied to design problems in the database area as well. The design and evaluation of new systems (or models) can be guided automatically at a very early stage of their development.

This paper, in essence, introduces, compares, and tries to integrate two such methods for database software development, analysis, and evaluation: The 'Vienna Development Method' (VDM) [1] as a more theoretically oriented semantic specification technique, and the 'Positional Set Notation' (PSN) [3] as the 'meta-language' for an automatic data model specification interpreter, the 'Data Model Processor' (DMP) [4]. The paper is divided into three major parts (chapters):

The second chapter introduces the basic semantic specification tools of both methods. It compares both meta-languages by applying them to a common object system, a subset of the relational data model (RDM) as specified in [9] and [5]. The basic result is that both methods have different objectives, and, thus, different advantages: PSN is oriented towards automatically interpretable specifications, VDM, on the other hand, provides the more complete and comprehensive specification meta-language.

The following chapters explore two approaches to combining the advantages of both methods:

In chapter 3, VDM is applied to specify the semantics of PSN in a completely formal way. The resulting formal model provides PSN and the specification language of the DMP with a concise and precise description of both their structures and operations. Based on this specification, a very detailed insight into the semantics of the meta-language can be obtained, an essential pre-condition for the use of the describing (meta-) language tools of any semantic specification method.

Chapter 4 describes the alternative approach of using the DMP directly as an interpreter for a subset of the formal semantic specification tools of VDM. The intended goal is to combine the advantages of a high-level, abstract semantic specification meta-language with the ease of analyzing its specifications by automatic means. It is shown in detail, to what extent PSN and the DMP commands can provide a machine executable interpreter for a substantial subset of VDM. Such an interpreter for high-level semantic models is especially well suited for simulating semantic specifications in the database area.

2. TWO APPROACHES TO DATA MODEL SEMANTICS SPECIFICATION

2.1. CONCEPTS OF A SEMANTIC SPECIFICATION METHOD

A semantic specification method provides the tools to describe the semantics of a given 'object system'. The set of all tools is called the 'meta-' or 'specification' language, the resulting description a 'semantic model' or a 'meta system'. Compared to the object system to be described, the meta system is expected to be considerably clearer, easier to understand, more concise, precise, and less ambiguous. This leads to certain requirements for the meta language. The appropriate choice of its tools and language constructs is the most important prerequisite for any specification technique. The set of tools provided should be, on one hand, as formal, concise and theoretically well based as necessary, and, on the other hand, as well understood and understandable for the intended user as possible. Experiences with natural language descriptions seem to indicate that formal ways of expressing semantics are more appropriate for that purpose than informal. (Formal specifications of complex systems, however, tend to be difficult to understand for untrained users.)

In any case, the meta-language tools should reflect the most important parts of the object system in a suitable way. Considering a data model as the system to be described, its main components can be characterized as

- a set of <u>objects</u> which may have a complex inner structure and different subcomponent relationships,

- a set of <u>operations</u> which are applicable to these kinds of objects, and (possibly)

- a collection of restrictive ('consistency' or 'integrity') <u>constraints</u> and conditions which have to be fulfilled by all 'well formed' objects and for all operations 'well applied' to their respective operands.

So, an appropriate meta language has to provide at least:

- a mechanism to specify the <u>structural</u> aspects of an object system with respect to both its subcomponents as well as its subcomponent relationships,

- a mechanism to express the semantics of the <u>operational</u> aspects of an object system with respect to both their abstract syntax (operation name, parameter types) as well as their semantic meaning, and

- ways to specify constraints which may apply to both the structural and the operation aspects of the object system.

In addition, the meta language should be based on a well defined set of primitives whose semantics can be derived from well known (preferably mathematical) objects. These primitives should be expressive enough to reflect the object system in an adequate way and the semantic model should be 'comprehensible' enough for either human or automatic use (or even both).

2.2. THE VIENNA DEVELOPMENT METHOD

The 'Vienna Development Method' (VDM) [1] is a semantic specification method based on the denotational approach. Besides several applications to programming language semantics, it has been successfully applied to semantic specifications in various database areas, ranging from a specific database language [7], to a complete database system [8], and a major data model [9].

VDM's corresponding meta-language, 'Meta IV', uses high-level, abstract, mathematical objects to 'denote' the semantic properties of an object system to be specified. An important aspect of the abstract 'meta' objects of Meta IV is that they express only a logical view of the structures, operations, and constraints of the object system without any implementational details. At the same time, they do not hide important structural aspects of the described system as some other semantic specification methods do. This makes a method like VDM especially well suited for specifications in the database area. Meta IV provides a very rich set of abstract modeling tools both for the specification of structures and operations as well as for constraints which may be imposed on them. The syntax of the meta-language tools was designed with special consideration to ease of understanding by users who are familiar with the (concrete) syntax of a common high-level programming language.

2.2.1 STRUCTURAL TOOLS

2.2.1.1 Elementary Objects

Meta IV provides (in principle) an unlimited set of elementary data types for the definition of semantic objects. These data types are well known from the usual programming languages: Boolean (value set BOOL), Integer and Real (also comprised of Numeral, value set NUM), and Quotation (for indivisible character strings, value set QUOT). The set TOKEN contains all those elementary objects whose inner structure is not considered any further in the formal model.

2.2.1.2 Structuring Mechanism

Additionally, the meta language provides a standard set of composite, higher-level abstract data types to describe the structural aspects of an object system. These types include:

- simple sets of elementary or higher-order elements,

- ordered tuples or lists of elementary or higher-order elements,

- labeled or unlabeled trees of elementary or higher-order objects,

- finite maps between (finite) sets of abstract objects, and

- total, partial, or bijectional functions between any abstract sets within the model.

Using these data types in an abstract VDM model, more complex meta-language structures are composed out of named, simpler ones in an (in general recursive) BNF-like style.

2.2.2 OPERATIONAL TOOLS

As VDM is based on the 'denotational' approach of formal semantics, the meaning of each object system's operation is defined in terms of a 'semantic meaning function' which represents the abstract 'denotation' of that particular operation. These so called 'elaboration functions' describe the meaning of each operation by specifying its effects on the semantic objects as given in the structural part of the formal model. In pure denotational semantics, the semantic meaning functions are represented by Lambda expressions. To improve the readability of the specifications, however, Meta IV provides alternative, but semantically equivalent means of writing certain classes of Lambda expressions. The alternative notations are chosen to be similar to well known constructs of modern high-level programming languages: they include constant and variable declarations, (meta) statements, conditional and iterative control structures, etc.

In addition, the usual operations of the higher-level abstract data structures may be used in the function specifications, provided their respective meanings are well defined in terms of set theory or first-order predicates.

2.2.3 CONSTRAINT MECHANISMS

For abstract objects as well as for the abstract operations, certain 'consistency constraints' may be formulated by use of the 'is-well-formed' predicates (i.e. functions) of the meta language Meta IV:

- 'static' consistency constraints, to restrict the classes of abstract objects to only the well formed ones, and

- 'dynamic' consistency constraints, to restrict the abstract operations to only the well applied ones, i.e. to those which are applied to well formed operands only.

2.3. THE POSITIONAL SET NOTATION

The 'Positional Set Notation' (PSN) [3] provides a powerful notation for specifying data modeling objects. PSN uses 'positional sets' (p-sets) as a unified way to define various aspects of data model structures formally. Furthermore, there exist a software system, the 'Positional Set Processor' (PSP) [4] which consists of about 40 powerful commands to define and manipulate PSN structures (i.e. p-sets). Thus, the specification language of the 'data model processor' (DMP) [5], i.e. PSN combined with the PSP commands, can be used as a well defined method to express data model semantics formally. The DMP provides an executable 'meta'-language which is based on p-sets, PSP commands, and some additional C-code extensions to specify behavioural properties which cannot be expressed completely by PSP operations. The main advantage of the formal semantic specification tools is their ability to be executed automatically on the DMP. Compared to VDM, however, the specification meta-language is somewhat more restricted, less abstract, and more difficult to understand and to use for an inexperienced user.

2.3.1 STRUCTURAL TOOLS

2.3.1.1 Elementary Objects

The basic elements of PSN are called 'atoms'. In this context, an atom is either a number (real or integer), a string, or 'null', as denoted by the special character '#'. Some character strings (namely 'indexes' or 'position identifiers') are constrained to begin with an alphabetic character.

2.3.1.2 Structuring Mechanism

In PSN, the basic mechanism to specify structures is the 'positional set' (p-set). P-sets are (indexed) sets of ordered pairs in which the first element of each pair is an atom or a positional set, the second, called the 'index' or 'position identifier', a (restricted - see above) character string, a number, or the special character '#'. So, the essence of PSN is the recursive definition of a p-set:
s = [x_i @ p_i . . .] where the x_i (elements) are either atoms or p-sets, and the p_i (position identifiers) are either atoms or the special character '#' - the null position identifier which denotes the index for each 'classical' set element. A pair x_i@p_i is called a 'duplex', i.e. each p-set consists of a (unordered) set of duplexes. It is demonstrated in [6] that p-sets are powerful enough to describe all structural properties of the main database models.

2.3.2 OPERATIONAL TOOLS

In PSN, the mechanism to specify operations is based on a set of 'primitive' or standard operations. They can be executed on a software system called the 'Positional Set Processor' (PSP) [4]. These 'PSP-commands' include the classical set operations, special operations to create, delete, and modify p-sets, so called 'sequence' operations, and auxiliary 'utility' operations. (See [4] for details.) Whenever there is no such (in general complex) application of PSP-commands to express the semantics of a data model operation completely, further semantic descriptions are added by use of 'concrete', plain C code. So all the semantics of an object systems operation is specified in terms of PSP operations, usually imbedded in some additional C code, i.e. all specifications are machine executable on the DMP.

2.3.3 CONSTRAINT MECHANISMS

The two possible kinds of consistency constraints may be specified in PSN as follows:

- structural constraints, i.e. restrictions to the structural contents of the p-sets are expressed using a special 'WHERE-clause' of the p-set 'CREATE' operation or the 'TEMPLATE' definition which define the build-up of new p-sets (See details in [6] and an example in 4.2.1.),

- operational constraints, i.e. restrictions to the data model's operations and their 'legal' operands, are expressed using predicates and/or Boolean expressions as provided by the programming language C.

So, PSN provides a meta language which is based on (indexed) p-sets, primitive PSP-operations, and C-code-extensions. As all these tools are formal and even machine executable, their semantics is well defined with respect to any software system which 'understands' p-sets, PSP-commands, and C-code. The Positional Set Processor was developed for that purpose.

2.4. AN EXAMPLE

In order to compare the two specification methods, we will now apply both of them to a common object system. As a small example we choose a subset of the relational data model as formally specified in two alternative ways in [9] and [6].

2.4.1 THE VDM SPECIFICATION

2.4.1.1 Structures

The abstract structures are specified in VDM in the 'semantic domains' of the formal model. In this example, we introduce the basic concepts of a relational database, relations, tuples, elementary values, and relation- and attribute-identifiers.

In the first line of the semantic domain definitions (1), we 'denote' the set of all possible relational databases (RDB) by an abstract VDM (meta) data structure 'map' which maps each relation identifier (from RELID) to its corresponding relation variable (from REL). Relation variables, in turn, are denoted by the abstract set of all tuples which they contain (2). Each abstract model of such a tuple (from TUPLE) maps its set of attribute identifiers (from ATTRID) to their respective values (3). Values may be taken from one of the three predefined VDM (meta) value sets: numeral (NUM, i.e. integer or real), Boolean (BOOL), or 'quotation' (QUOT) which represents the set of undivisible character strings (4). Finally, both kinds of identifier sets are not considered any further in this semantic model, therefore being regarded as just 'tokens' (from the predefined VDM set TOKEN, see lines 5 and 6).

SEMANTIC DOMAINS

```
RDB     =   (RELID ---> REL)             (1)
REL     =    TUPLE - Set                 (2)
TUPLE   =   (ATTRID ---> VAL )           (3)
VAL     =    NUM | BOOL | QUOT           (4)
RELID   =    TOKEN                       (5)
ATTRID  =    TOKEN                       (6)
```

As the consistency or integrity constraints can be classified as either structural (or 'static') or operational (or 'dynamic'), we will now present each set together with the corresponding (structural or operational) part of the formal model.

STATIC CONSISTENCY CONSTRAINTS

In VDM, the semantic domains define sets of abstract objects denoting the structural components of the system to be described. The static consistency constraints restrict the sets of all possible abstract objects to only their 'well formed' elements. All consistency constraints are expressed as predicates (i.e. Boolean functions), mapping the well formed objects to true, all others to false. As all abstract VDM functions, the predicates have their respective types listed in the last line of the corresponding formal function specification.

So, in our example, it is stated as a static consistency constraint in [9] that all tuples within each database relation have the same set of attribute identifiers (i.e. their denotations have the same domain, dom). For instance, this constraint (and, maybe, further additional constraints) is formally specified by the following predicate. It maps any relational database (from RDB) to true (from BOOL) if it is 'well formed', and to false otherwise:

```
is-wf-RDB (rdb) =

    ( ∀  relid  ∈  dom rdb )
        (  ∀  tuple1, tuple2  ∈  rdb (relid) )
            ( dom tuple1  =  dom tuple2 )
    and    ...  (further constraints)  ...

type :   RDB   --->  BOOL
```

2.4.1.2 Operations

In VDM, the abstract structure of the operations is specified in the 'syntactic domain' part of the formal model. As an example, we describe the 'insert' operation (from the abstract set 'Insert') which inserts a single tuple into a relation. First, the operation's abstract syntactical structure is defined: The insert operation is based upon an identifier (from RELID) for the relation to be altered, and on the new tuple (from Newtuple) which is to be inserted into the relation. This new tuple is, of course, denoted by an element of the abstract set TUPLE as defined in the semantic domains above.

SYNTACTIC DOMAINS

```
Insert      ::  RELID  Newtuple
Newtuple    =   TUPLE
```

Similar to the static consistency constraints, the abstract denotations for the operations (described in the syntactic domains) are usually constrained too. These operational or dynamic restrictions are defined in the 'dynamic consistency constraints' which state whether an operation is 'well applied' to well formed operands or not.

For instance, a new tuple may only be inserted into a relation whose declaration already exist in the database (1). Furthermore, tuples in the content of such a selected relation variable must have the same set of attribute identifiers as the new tuple to be inserted (2).

These constraints, again, are expressed in VDM by a Boolean function (is-wf-Insert) which maps an abstract representation of the insert operation (from Insert) together with a model of the database (from RDB) to a Boolean value.

DYNAMIC CONSISTENCY CONSTRAINTS

```
    is-wf-Insert (insert) (rdb) =

              s-RELID(insert)  ∈  dom rdb                          (1)
         and  ( ∀ tuple  ∈  rdb (s-RELID (insert)) )
                 ( dom tuple = dom s-Newtuple (insert) )           (2)

    type : Insert ----> ( RDB ----> BOOL )
```

The semantic meaning of each operation is denoted by a function in the 'elaboration functions' part of the formal model. This function describes the transition from one database state (from RDB) to a new, altered one. Its type is given by an abstract map which relates a database state change (from the abstract set of maps (RDB--->RDB)) to the execution of each given insert operation (from Insert). So, for each insert operation and an actual database state, the elaboration functions yields a new database state.

ELABORATION FUNCTIONS

```
    elab-Insert (insert) (rdb) =

         let  mk-Insert (relid,newtuple) = insert  in     (1)
         let  set1  be  rdb (relid) ,                     (2)
              set2  be  { newtuple }      in              (3)
         let  newset  be  set1 u set2     in              (4)
              rdb  +  [ relid ---> newset ]               (5)

    type :  Insert  --->  ( RDB ---> RDB )
```

The specification of the elaboration function for the insert operation (eval-Insert(..)) starts with making available the two parts of its input parameter ('relid' and 'newtuple') for the scope of the function specification (1). Then, two set-constants are defined to contain the old relation value and the new tuple to be inserted into it (2,3). A third set constant is declared to contain the union of the first two sets (4). The result of this elaboration function is the denotation of the altered database state, denoted by a map, rdb, which is updated with respect to one element, relid, of its domain (5).

2.4.2 THE PSN SPECIFICATION

The whole interactive software environment to define, test, and evaluate data models specified in PSN is the 'data model processor' (DMP) [5]. The DMP is divided into several parts, describing the different human roles which a user may play in the process of specifying, initializing, and evaluating a data model and its applications. The most important part he has to play is that of a 'data model definer'. In this role, a user defines the semantics of an intended data model by providing a complete specification of its semantics using PSN.

2.4.2.1 Structures

In a data model specification in the DMP framework, the data model definer first uses PSN to define the structures that can be expressed and manipulated in the data model to be described. All abstract data structures are defined in the 'p-set definition' part of the formal model. In its 'concept definition section', the new concepts are introduced together with the p-sets which represent them in the model.

We demonstrate the PSN specification method for data models by applying it to the same example as above: In [6], a relational database is viewed (in part) as a set of different relation occurrences. So, in this example, the new concept 'RELATIONS' is declared as a PSP 'occurrence-structure' (using the DMP command REP-OCC) and will be represented by the p-set 'REL-OCC'. (In addition there might also be a 'definition-structure', declared by REP-DEF, for type definitions. This concept, however, will not be mentioned in this example.)

CONCEPT DEFINITION SECTION

```
      . . .
      ---> REP-OCC  RELATIONS  WITH  REL-OCC
      . . .
```

In the following 'p-set definition section', the p-sets declared in the previous section are defined in detail using other DMP operations. First, in a command file (1-3), a range declaration (RG) associates all later used range variables with their respective p-sets (2). Then, the 'TEMPLATE' command provides a description for each p-set without actually enumerating its elements explicitly (4-10): a p-set skeleton lists the configuration of position identifiers (4); the following 'WHERE' clause further constrains the set of all possible duplexes which may appear in such a p-set (5-10).

Again, we concentrate on the specification of relation occurrences (REL-OCC) only:

P-SET DEFINITION SECTION

```
   ---> BEGIN  <range-definition-command-file>                 (1)
           -->  RG RO IS REL-OCC                               (2)
           -->  END <range-definition-command-file>            (3)

   ---> TEMPLATE  REL-OCC = [[RN@Name,
                              RR-TPL@Relation]# ...]           (4)
        WHERE                                                  (5)
           ISA -C RN,                                          (6)
           CD REL-OCC = CD (CR WITH (RO.Name)),                (7)
           TEMPLATE  RR-TPL = [[V@P(J)..J]@#...]               (8)
              WHERE                                            (9)
                 ISIN P(J) ATTF(RN)                           (10)
                 . . .
```

STRUCTURAL CONSTRAINTS

In PSN, the structural consistency constraints are part of the p-set definition. In the example given above, for instance, it is stated that all relation names (RN) have to be of type 'character' (-C) (6), relation names have to be unique within the set of all relation occurrences (7), and the tuple attribute identifiers have to match those of the relation's type definition (as derivable by an auxiliary function ATTF) (8-10).

2.4.2.2 Operations

In the DMP framework, the semantic specification of the operations is based on the set of executable PSP commands. In the environment of a C program which emulates the specified operations, the execution of a PSP command is invoked by a procedure call to the PSP (expsp(...)) - similar to a system call.

In this example, one temporary p-set is created (CR) for the argument relation's value (3), another one for the tuple to be inserted (EN) (9); then, the union (UN) of both p-sets is established as the new relation value (10). Prior to its use, each new range (RG) variable (declared in 2,5,7) is released (RL) from any prior association with another p-set (1,4,6).

```
insert-T (REL,NEWTUP)
char *REL, *NEWTUP;
{
  . . .
   expsp ("RL X1");                                        (1)
   expsp ("RG X1 IS REL-OCC");                             (2)
   expsp(stringf(buff1,"CR TMP1 WITH \"(X1.ALL)\"
                 WHERE \"(x1.Name = '%s')\"",REL));        (3)
   expsp ("RL Q");                                         (4)
   expsp ("RG Q IS TMP1");                                 (5)
   expsp ("RL R");                                         (6)
   expsp ("RG R IS Q.Relation");                           (7)
   expsp ("CR TMP2 WITH \"(R.ALL)\"");                     (8)
   expsp(stringf(buff2("EN \"[[%s]@#]\" NEWTUP",
                                      NEWTUP));            (9)
   expsp ("UN TMP2 NEWTUP INTO TMP3");                    (10)
  . . .
}
```

2.5 COMPARISON OF THE TWO SPECIFICATION METHODS

Most of the differences between VDM and PSN stem from the different purposes for which each of them was originally designed.

VDM is a specification method which is intended to be a tool for a more systematic approach to (in essence) manual development of software. It uses a meta-language incorporating the modeling power of first-order logic to ease the appropriate modeling of more complex object systems (such as programming languages as well as database models). Its formal tools are very rich, high-level, and applicable in a flexible way. VDM emphasizes comprehensiveness of its abstract semantic models for a broader set of users. VDM's semantic models, however, are not machine processable.

PSN is designed as a basic, but yet sufficiently powerful formal notation to support data modeling. It is based on a small set of well defined mathematical concepts. Its meta-language does not emphasize comprehensiveness for the (mathematically inclined) human reader. The notation is simple, unambiguous, and straightforward for modeling the structural aspects of a (data model) object system. Abstract formal PSN models are machine processable for a computer based interpreter, the DMP.

The specification of the operational/behavioural aspects of an object system is based on denotational semantics in VDM. That is, the specification is based on sound mathematical concepts and the semantics of the meta-language is well defined. Meta IV is powerful enough to express completely all semantic aspects of any object system (with the exception of concurrency).

In the PSP, the specification of the operational aspects of an object system relies on an extensive (but not totally comprehensive) set of PSP commands. All of them are implemented to be machine executable. However, the semantic meaning of this kind of meta-language is basically defined by its implementation only. Some parts of the operational aspects of a PSN-based model still have to be expressed in terms of their implementation in C code. Whether the PSN primitives could be extended to avoid the necessity of C-code augmentations is still an open and debatable question.

In VDM however, the full and extensive use of the meta-language may lead to many different, complex, or even inconsistent abstract semantic models. All analyses, tests, and interpretations have to be performed 'manually', because aids to any kind of automatic interpretation of the model do not exist - and may be subject to human error. (As mentioned above, the sheer volume of a complex specification can make it quite difficult to ensure its consistency.)

The two specification methods compared in this report have different objectives and, thus, different properties. An important objective of PSN is to provide automatically treatable formalizations of data model semantics. But this, on the other hand, leads to some lack of richness and formally sound foundations of the meta-language (especially with respect to its ability to specify operations). Contrarily, VDM emphasizes a rich, powerful and more easily understood meta-language. But Meta IV has no automatic aids to analyze and/or interpret the semantic models (although they tend to be rather complex when specifying non-trivial object systems).

In order to overcome some of the deficiencies of the two methods in consideration, two major solutions arise:

The first one would be to provide the PSN meta-language (i.e. the PSP commands) with a semantically sound formal definition. This could be expressed using any formal semantic specification technique, an appropriate one being, e.g., VDM. This approach is further elaborated in chapter 3. In addition, the meta-language should be extended in order to avoid the use of C-code in its behavioural specifications.

The other alternative would be to make VDM's abstract models treatable by some kind of automatic analyzer and/or interpreter. This seems to be more difficult to apply to the whole extent of meta-language features in Meta IV. But a semantic specification that uses the full set of 'Meta IV' language constructs could be transformed into one that uses only a certain subset. It should then be possible to interpret the constructs of such a (slightly restricted) specification automatically using an appropriate processor. Choosing the PSP for this task would mean integrating the advantages of both methods in a quite natural way. The second approach is further addressed in chapter 4.

3. FORMAL SEMANTICS OF THE DATA MODEL PROCESSOR

3.1. INTRODUCTION TO THE FORMAL MODEL

This chapter provides a completely formal semantic specification of PSN and major parts of the corresponding PSP operations as described in [5] and [6]. The 'object system' to be modeled here consists of the structured objects (p-sets) as defined in PSN, accompanied by a set of (PSP) commmands to manipulate these structures.

As an appropriate semantic specification method, VDM is choosen. According to VDM, the formal semantic model of PSN is divided into the following major components:

- the '<u>semantic domain</u>' definitions (3.2.1) specify the denotations for all structural aspects of PSN upon which the operations of the PSP are based,

- the '<u>syntactic domain</u>' definitions (3.3.1) list the abstract syntax of all operations (i.e. PSP commands) and describe the structure of their respective operands,

- the '<u>elaboration functions</u>' (3.3.3) specify the meaning of the operations; their semantics is denoted by semantic functions on the abstract sets of structure denotations as defined in 3.2.1 and 3.2.2, and

- the '<u>consistency constraint</u>' definitions state additional restrictions on both the structural (or 'static') (3.2.3) and the operational (or 'dynamic') (3.3.2) aspects of the model.

In the following subsections, short informal introductions precede the formal specifications of each part of the semantic model. These introductions are not meant to be part of the semantic model, but should rather give some explanatory hints for readers who are less familiar with the meta-language of VDM.

In the informal introductions, newly introduced concepts are abbreviated (in brackets) in the same way as their denotations are identified in the formal specifications.

3.2. STRUCTURE SPECIFICATION

3.2.1 ABSTRACT SYNTAX

The PSP provides an interpreter for a set of high-level operations which can be applied to p-sets. The meaning of the 'PSP commands' is based upon two auxiliary structures which have to be specified in a formal semantic model:

- A 'range variable table' (RVTABLE) contains all declared range variable identifiers (RVID), together with their corresponding ranges (RANGE). These ranges may be defined either by a positional set identifier, or by a 'term' (TERM), i.e. a selected element of another range variable's range.

- A 'freeze table' (FRTABLE) contains at all times the actual bindings for each range variable to one particular duplex out of its respective range.

All identifiers used in this model are regarded as being just 'tokens' which have no further semantic content.

<u>S E M A N T I C   D O M A I N S</u>

```
PSN       ::    PSETS  RVTABLE  FRTABLE

PSETS     =    (PSID ---> PSET)

PSET      =     DUPLEX - Set

DUPLEX    ::    ELEM  POSID

ELEM      =     ATOM  |  PSET
POSID     =     #  |  NUM  |  CHAR

ATOM      =     NUM  |  CHAR
CHAR      =     QUOT+

RVTABLE   =    (RVID ---> RANGE)

RANGE     =     PSID  |  TERM
TERM      ::    RVID  POSID

FRTABLE   =    (RVID ---> DUPLEX)

PSID      =     TOKEN
POSID     =     TOKEN
RVID      =     TOKEN
```

3.2.2 THE GLOBAL STATE

Later on, we specify the constraints and some PSP operations for the PSN structures defined above. Formally, all their semantics is 'denoted' by semantic functions based on these structures. Thus, nearly all semantic functions would have to contain the PSN structure definitions as part of their parameters. To make these function definitions more readable, VDM provides a way to declare the main structural parts of a semantic model as 'global' (meta-) variables, i.e. to make them globally 'accessible' for each semantic function. In our case, for example, the 'global state' for a PSN model consists of three components which are declared and initialized in the following way:

G L O B A L   S T A T E

$\underline{\text{dcl}}$ $\underline{\text{PS}}$:= [...] $\underline{\text{type}}$ PSETS

$\underline{\text{dcl}}$ $\underline{\text{RV}}$:= [...] $\underline{\text{type}}$ RVTABLE

$\underline{\text{dcl}}$ $\underline{\text{FR}}$:= [...] $\underline{\text{type}}$ FRTABLE

All global variables are initialized with the component values of the corresponding semantic domain objects. Thus, for any given psn ∈ PSN, the global variables refer to the contents, $\underline{c}$, of its respective subcomponents at any time:

$\underline{c}$ $\underline{\text{PS}}$ = s-PSETS (psn) ,
$\underline{c}$ $\underline{\text{RV}}$ = s-RVTABLE (psn) , and
$\underline{c}$ $\underline{\text{FR}}$ = s-FRTABLE (psn) .

The whole state is referred to as 'S' in the 'type' declarations of the following semantic functions.

3.2.3 STRUCTURAL CONSTRAINTS

In addition to the structural properties defined above, each 'well formed' PSN model is constrained (at least) by the following requirements:

1. p-sets and range variables have to have different identifiers,

2. all 'frozen' range variables from the freeze table have to be declared in the range variable table,

3. all p-sets have to be 'well formed', i.e. all their position identifiers have to begin with a letter,

4. the range variable table has to be 'well formed', i.e. it may contain only well defined p-sets as the range of each range variable, and

5. the freeze table has to be 'well defined' too, i.e. each range variable can only be frozen to a duplex from its correct respective range.

These constraints are specified by a Boolean function (predicate), is-wf-PSN, with auxiliary functions used whenever the readability can be improved. (The numbers in brackets refer to the list above.)

S T A T I C   C O N S I S T E N C Y   C O N S T R A I N T S

```
is-wf-PSN ()  =

            dom  c PS  n  dom  c RV  =  {}           (1)

        and dom  c FR  <  dom  c RV                  (2)

        and ( ∀  pset  ∈  rng  c PS )                (3)
                  ( is-wf-PSET (pset) )

        and is-wf-RVTABLE ()                         (4)

        and is-wf-FRTABLE ()                         (5)

    type : S  ---->  BOOL
```

3.2.4 AUXILIARY FUNCTIONS

Auxiliary functions are used either to specify common parts of different functions of the formal model, or to express certain properties in more detail at a separate place to improve the readability of the specifications.

Here, the auxiliary functions for the specification of the static consistency constraints first define in detail which p-sets, range-variable-, and freeze-tables are well formed:

A p-set is well formed, iff it is structured as described in the semantic domains, and, in addition, the position identifiers of all duplexes in the p-set begin with a letter, if they are character strings:

```
is-wf-PSET (pset)  =

   ( ∀  duplex  ∈  pset )
   ( let  posid  be  s-POSID (duplex)  in
          is-CHAR (posid)  ===>
                   posid [1]  ∈  {A, ... ,Z} )

type :  PSET  ---->  BOOL
```

A range-variable-table is well formed, iff for all declared range variable identifiers their corresponding ranges are well formed. (Note: As the parameter, RV, to this function is part of the global state, S, the function specified here has no explicit parameters.)

```
is-wf-RVTABLE ()  =

   ( ∀  rvid  ∈  dom c RV )
     ( is-wf-range (c RV (rvid)) )

type :  S  ---->  BOOL
```

What is meant by a well formed 'range' is then specified formally by another auxiliary function which states the well-formedness conditions for both possible kinds of ranges (PSID or TERM):

- either the position identifier is an actually declared p-set which is part of the content of the global state variable PS, or,

- if the range is defined in terms of another range variable identifier, the position identifier selects an actually existing duplex out of that particular range variable's range (which is determined by another auxiliary function, get-range).

```
is-wf-range (range)  =

   cases  range  of :

   (mk-PSID (psid)  --------->  psid ∈  dom c PS ,

    mk-TERM (rvid,posid) ---->  rvid ∈  dom c RV
          and ( ∀ duplex ∈ get-range (rvid) )
               ( is-PSET (s-ELEM (duplex))
                 and  ( ∃ duplex' ∈ s-ELEM (duplex) )
                         ( s-POSID (duplex') = posid ) ) )

type :  RANGE  ---->  BOOL
```

The freeze-table is well formed, iff each range variable identifier in its domain is 'frozen' to a duplex from the corresponding range.

```
is-wf-FRTABLE ()  =

   ( ∀  rvid  ∈  dom c FR )
     ( c FR (rvid)  ∈  get-range (rvid) )

type :  S  ---->  BOOL
```

The next two auxiliary functions formally define the range of a range variable identifier and specify how to select a set of duplexes from a p-set identified by a given position identifier.

The corresponding range for a range variable identifier is a set of duplexes which is determined by the auxiliary function 'get-range':

```
get-range (rvid)  =

   cases  c RV (rvid)  of :

   (mk-PSID (psid)  ---------->  c PS (psid) ,

    mk-TERM (rvid',posid)  --->  { duplex |
                                duplex   ∈  dot (duplex',posid)
                             and duplex'  ∈  get-range (rvid') } )

pre   :  rvid ∈  dom c RV  and  is-wf-range (c RV (rvid))
type  :  RVID  ---->  PSET
```

Given a position identifer, the auxiliary function 'dot' selects the corresponding duplex from a p-set. Because, in general, position identifiers in a p-set are not necessarily distinct, the result is a set of selected duplexes, i.e. a p-set.

```
dot (pset,posid)  =

    let duplex  ∈  pset  be  s.t.
        s-POSID (duplex) = posid  in
        s-ELEM (duplex)

pre  :  ( ∃ duplex ∈ pset )
             ( s-POSID (duplex) = posid )
type :  PSET  POSID  ---->  PSET
```

Both of the last two functions are applied correctly only if some additional (pre-) conditions on their input parameters are satisfied. These pre-conditions are formally stated in the 'pre' predicates at the end of the function specifications. Each environment which uses any of these functions has to make sure that all pre-conditions are satisfied prior to a function application.

3.3. OPERATION SPECIFICATION

3.3.1 ABSTRACT SYNTAX

The PSP is a software tool for interpreting and manipulating p-sets. With the PSP commands, it provides a set of powerful operations which can be applied to the PSN structures as defined above.

In this subsection, the abstract syntax of the PSP operations is analyzed, i.e. their names are declared, and the structure of their respective operands is described in an abstract way without considering any details of the syntactic notion.

Not all PSP commands will be considered in the semantic model. As an example, only some of the more important operations are defined formally. Those PSP commands (Psp-cmd) specified here include: the 'range-variable' declaration (Range_dcl), the 'release' command for range variables (Release), the command to 'enter' a p-set expression into a p-set (Enter), the 'delete' (Delete), 'freeze' (Freeze) and 'thaw' (Thaw) commands for p-sets, and the more complex 'create' (Create) command to build up and store new p-sets.

P-set expressions (Ps-expr) describe all possible ways of creating new sets of duplexes (i.e. p-sets) from existing structures: from a p-set-identifier (PSID), a whole p-set (PSET), a classical set (C-set), a union of two other p-set expressions (Un-expr), or a p-set 'create' expression (Ps-cr-expr), etc.

S Y N T A C T I C   D O M A I N S

```
Psp-cmd     =    Range-dcl | Release | Enter |
                 Delete | Freeze | Thaw | Create | . . .

Range-dcl   ::   RVID  RANGE

Release     ::   RVID

Enter       ::   PSID  Ps-expr

Delete      ::   PSID

Freeze      ::   RVID Cond

Cond        =    (DUPLEX ---> BOOL)

Thaw        ::   RVID

Create      ::   PSID  Ps-cr-expr

Ps-cr-expr  ::   A-list  Cond

A-list      =    Attribute - Set
Attribute   =    TERM | Attr-asgn
Attr-asgn   ::   POSID  Phrase

Phrase      =    ATOM | Arth-expr | TERM
Arth-expr   =    . . .

Ps-expr     =    PSID | PSET | DUPLEX | C-set
                      | Un-expr | Ps-cr-expr | . . .

C-set       =    ELEM - Set

Un-expr     ::   Ps-expr  Ps-expr
```

3.3.2 OPERATIONAL CONSTRAINTS

The operations are basically constrained by the kinds of arguments they can take. In the previous subsection, some of those constraints are expressed through the structural definition of the operations's operands. But there are further 'operational constraints', e.g.: not only must the first argument of the 'Range' command be a range variable identifier (rvid), but also must all mentioned range variable identifiers be declared previously, i.e. be part of the domain of the range variable table referred to by $\overline{\underline{RV}}$. Similarly, the second argument (range) of this command is restricted with respect to the structure definitions given in the 'semantic domains', etc.

In the same way as the structural constraints, operational constraints are defined as Boolean functions, i.e. predicates, over the set of abstract operations as listed in the syntactic domains. Here again, only some operational, or 'dynamic' constraints will be defined exemplarily.

D Y N A M I C   C O N S T R A I N T S

The dynamic consistency constraints check whether the operations are applied to well formed operands. Here, the whole (Boolean) function to check the well-formedness of the PSP commands is divided into different cases according to all listed alternatives for the operations. An (is-well-formed) predicate is specified for each PSP command which is described in the syntactic domains:

```
   is-wf-Psp-cmd (psp-cmd) =

     cases psp-cmd of:

     (mk-Range (rvid,range) -----> is-wf-range (range)
                                   and rvid ∈ dom c RV ,

      mk-Release (rvid) ---------> rvid ∈ dom c RV ,

      mk-Enter (psid,ps-expr) ---> is-wf-Ps-expr (ps-expr)
                                   and psid ∈ dom c PS ,

      mk-Delete (rvid) ----------> rvid ∈ dom c RV ,

      mk-Freeze (rvid,cond) -----> rvid ∈ dom c RV
                and ( ∀ duplex ∈ get-range (rvid) )
                               ( duplex ∈ dom cond ) ,
   mk-Thaw (rvid) -------------> rvid ∈  dom c RV ,

   mk-Create (psid,ps-cr-expr) ----->
        let  mk-Ps-cr-expr (a-list,cond)
                                 be  ps-cr-expr  in
                  psid ∈  dom c PS
             and  is-wf-Ps-expr (ps-cr-expr)
             and  ( ∀ duplex ∈ max-pset (a-list) )
                      ( duplex  ∈  dom cond ) ,
  . . .                                         )

type :  Psp-cmd  ---->  BOOL
```

The next auxiliary function defines which p-set expressions are well formed. It is mainly used in the previous specification of the well formed operations (operands). Note that no additional restrictions are imposed on p-set expressions from the abstract sets PSET, DUPLEX, or C-set. Note also, that the predicate for the p-set 'create' expression, ps-cr-expr, makes sure that the condition, cond, in this expression is defined for all possible duplexes which may conform to its attribute list. This 'maximum' p-set conforming to the specified attribute list can be derived by the auxiliary function 'max-pset' which is formally defined in 3.3.4.

```
is-wf-Ps-expr (ps-expr) =

  cases ps-expr of :

  (mk-PSID (psid) -----------> psid ∈  dom c PS ,

   mk-PSET () ---------------> true ,

   mk-DUPLEX () -------------> true ,

   mk-C-set () --------------> true ,

   mk-Un-expr (pse1,pse2) ---> is-wf-Ps-expr (pse1)
                           and is-wf-Ps-expr (pse2) ,

   mk-Ps-cr-expr (ps-cr-expr) ------------>
      let mk-Ps-cr-expr (a-list,cond) be ps-cr-expr  in
          ( ∀ duplex ∈ max-pset (a-list) )
              ( duplex  ∈  dom cond ) ,

  . . .                                 )

type :  Ps-expr  ---->  BOOL
```

3.3.3 OPERATION SEMANTICS

In the last part of the semantic model for PSN, we finally define the semantics of the operations listed in the syntactic domains. The semantic meaning of each operation is denoted by an 'interpretation function' which describes the mapping of one instance of the global state (before the operation's execution) to a new one (after the operation's execution).

Thus, the interpretation function for PSP-commands specifies a state change for each different PSP operation. In case the operation is a range variable declaration/release, the global state component RV is updated with respect to that particular range variable. The meaning of entering/deleting a p-set in the model is denoted by a change of the global state component PS.

The semantic description of the 'freeze' command is divided into two different alternatives. In the first case, the range variable to be frozen (according to a given condition, cond) is not defined in terms of another range variable identifier. Then, a duplex satisfying the condition is chosen out of its respective range, and the global FR state component is updated with respect to that range variable identifier. The other alternative is that the range of the argument range variable identifier is defined by (in general) a list of 'ancestor' range variable identifiers (via 'dot' selection). In this case, the list is derived (ancestorl), all corresponding duplexes are chosen (duplexl), and then the FR-table is updated with respect to all ancestor range variable identifiers.

The 'thaw' command releases a given range variable identifier together with the elements of its list of 'descendent' range variable identifiers (derived by the auxiliary function 'descendents') from the FR-table in the global state. In the denotation of the 'create' command, the p-set expression, cr-ps-expr (as specified later) is evaluated first, and then the result is entered into the set of all defined p-sets referred to by PS.

The interpretation of each PSP command results in a global state change.

E L A B O R A T I O N   F U N C T I O N S

```
int-Psp-cmd (psp-cmd) =

  cases psp-cmd of :

  (mk-Range (rvid,range) -------> RV :=
                                  c RV + [rvid --> range] ,

   mk-Release (rvid) -----------> RV := c RV \ {rvid} ,

   mk-Enter (psid,ps-expr) ----->
                    let pset be eval-Ps-expr (pset)  in
                        PS := c PS + [psid --> pset] ,

   mk-Delete (psid) ------------> PS := c PS \ {psid} ,

   mk-Freeze (rvid,cond) ------->

   if parent (rvid) = undefined

      then let duplex ∈ get-range (rvid)
                        be s.t. cond (duplex)  in

           FR := c FR + [rvid --> duplex] ,

      else let ancestorl be ancestors (rvid)  in
           let duplexl be < dx[i] |
                   dx[i] ∈ get-range (ancestorl [i])
               and s-ELEM (dx[i]) ∈ dot (s-ELEM (dx[i-1]),
                          s-POSID (c RV (ancestor [i])))
               and cond (s-ELEM (dx[n]))
               and 2 ≤ i ≤ n=len ancestorl >  in

           FR := c FR + [rvid [i] --> duplexl [i] |
                         1 ≤ i ≤ n  and
                         rvid [i] = ancestorl [i]  ] ,

   mk-Thaw (rvid) ------------------>
           let descendentl be descendents (rvid)  in
               FR := c FR \ elems descendentl  ,

   mk-Create (psid,ps-cr-expr) ----->
           let p-set be eval-Ps-expr (ps-cr-expr)  in
               FR := c FR + [ psid --> p-set ]  ,

  . . .                                             )

type :  Psp-cmd  ---->  ( S ----> S )
```

The evaluation of all different kinds of expressions listed in the syntactic domains is denoted by 'evaluation functions' which map an expression and an actual instance of the state into a particular (in general complex) value. Here, we concentrate on the evaluation of p-set expressions only.

If a p-set expression is given by a p-set identifier, it is evaluated by deriving its corresponding p-set from the global state component PS. If a p-set is given, that p-set is simply returned. A duplex is evaluated as a singleton p-set that contains the duplex, and a classical set is mapped into a p-set of duplexes with null position identifiers. A 'union' command creates a p-set from the union of its evaluated argument p-set expressions. A p-set 'create' expression first creates the set of all possible duplexes conforming to the given attribute list (using the auxiliary function 'max-pset'), and then selects only those duplexes which satisfy the specified condition.

The evaluation of each p-set expression returns a p-set.

```
eval-Ps-expr (ps-expr) =

  cases  ps-expr  of :

  (mk-PSID (psid) -------------->  c PS (psid) ,

   mk-PSET (pset) -------------->  pset ,

   mk-DUPLEX (duplex) ---------->  { duplex } ,

   mk-C-set (c-set) ------------>  { mk-DUPLEX (elem,#) |
                                          elem ∈ c-set } ,

   mk-Un-expr (pset1,pset2) ---------->

               let p-set1 be eval-Ps-expr (pset1)   ,
               let p-set2 be eval-Ps-expr (pset2)  in
                   pset1 u pset2 ,

   mk-Ps-cr-expr (a-list,cond) ------>

               let m-pset be max-pset (a-list)  in
                   select (m-pset,cond) ,

  . . .                                         )

type :   Ps-expr   ---->  PSET
```

3.3.4 AUXILIARY FUNCTIONS

The first three auxiliary functions for the specification of the operations' semantics formally describe what is meant by 'parent', 'ancestors', and 'descendents' of a range variable identifier.

```
parent (rvid)  =

   if  is-PSID (c RV (rvid))
       then  undefined
       else  s-RVID (RV (rvid))

pre  :  rvid  ∈  dom c RV
type :  RVID  RVTABLE  ---->  RVID


ancestors (rvid) =

   < rvid [i] | parent (rvid [1]) = undefined
                and  rvid [i] = parent (rvid [i+1])
                and  rvid [n] = rvid  and 1 ≤ i ≤ n >

pre  :  rvid  ∈  dom c RV
type :  RVID  ---->  RVID+


descendents (rvid) =

   < rvid [i] | rvid [1] = rvid
                and  rvid [i] = parent (rvid [i+1])
                and  is-PSID (c RV (rvid [n]))
                and  1 ≤ i ≤ n  >

pre  :  rvid  ∈  dom c RV
type :  RVID  ---->  RVID+
```

The auxiliary function 'max-pset' derives for any given attribute list (from A-list) the (maximum) set of all possible duplexes which conform to that particular attribute list. It does this by first determining the ancestor 'chains' for all dependent and independent range variable identifiers. (A range variable identifier is regarded to be 'independent' if it is not defined in terms of any other range variable identifier.) Then a 'product'-set (similar to a relational 'join') is built up for all independent range variable identifiers. This set is finally 'projected' (and/or augmented) according to the given attribute list.

```
max-pset (a-list) =

  let rvid-chains  be
          get-rvid-chains (a-list)  in
  let independent-rvids  be  { rvidl [1] |
          rvidl ∈  elems rvid-chains }  in
  let product-pset  be
          product (independent-rvids)  in

   project (product-pset,a-list)

type :  A-list  ---->  PSET
```

The following auxiliary functions specify in detail the semantics of the mentioned steps in the previous semantic funcion.

```
get-rvid-chains (a-list) =

  let  rvids  be { rvid |  attr ∈ a-list  and
      (   ( is-TERM (attr) ==> rvid = s-RVID (attr) )
       or ( is-Attr-asgn (attr) ==>
            ( is-TERM (s-Phrase (attr))  and
              rvid = s-RVID (s-Phrase (attr)) ) ) ) }  in

  let  independent-rvids be { rvid | rvid ∈ rvids  and
       not ( ∃ rvid' ∈ rvids )
       ( rvid' ∈  elems ancestors (rvid) ) }  in

  { rvidl | rvidl [1] ∈ independent-rvids
          and rvidl [i+1] ∈ elems ancestors (rvidl [i])
          and ( ∀ rvid ∈ rvids ) (∃ rvidl, j)
                  ( rvidl [j] = rvid )
          and    1 ≤ i,j ≤  len rvidl }

type  :  A-list  ---->  ( RVID+ ) - Set


product (rvids)  =

  { mk-DUPLEX (elem,#) | elem = mk-PSET (pset)
    and  pset = { mk-DUPLEX (elem',rvid) |
                  elem' ∈ get-range (rvid)
                  and rvid ∈ rvids        } }

type :  RVID - Set  ---->  PSET
```

```
project (pset,a-list) =

  { mk-DUPLEX (elem,posid) | duplex ∈ pset   and
                             attr ∈ a-list   and

  cases attr of :

  (mk-TERM (rvid,posid') -------------->

      posid = posid'  and
      elem ∈ dot (dot (s-ELEM (duplex), rvid), posid) ,

   mk-Attr-asgn (posid',phrase) ------->

      posid = posid'  and

      cases phrase of :

      (mk-ATOM (a) ------------> elem = a ,

       mk-Arith-expr (ae) -----> elem =
                                 eval-Arith-expr (ae) ,

       mk-TERM (rvid,posid") --> elem ∈ dot (dot (

                 s-ELEM (duplex),rvid), posid") ) )   }

type :  PSET  A-list   ---->  PSET


select (p-set,cond) =

  { duplex | duplex ∈ p-set
             and  cond (duplex) }

type :  PSET  Cond   ---->  PSET
```

The evaluation of arithmetic expressions will not be specified in this context. For all possible kinds of arithmetic expressions it yields an atom.

```
eval-Arith-expr (arith-expr) =

  . . .

type :  Arith-expr   ---->  ATOM
```

4. INTERPRETING DATA MODEL SPECIFICATIONS

4.1. OVERVIEW

The main goal of this chapter is to combine the advantages of both VDM and PSN in a different way as proposed above. This can either be looked at as making VDM's abstract models treatable by some kind of automatic interpreter, or, from the other side, as providing PSN with a more abstract, flexible, and more readable meta-language on top of PSN and its already implemented PSP commands. Then, the database model specifications can be expressed in an abstract, high-level, and convenient way, and, on the other hand, the specified systems can be automatically emulated for machine supported testing and analysis at a very early stage of the design process.

The chapter proceeds with an outline of the process of mapping a slightly restricted, formal VDM model into a corresponding representation in terms of PSN structures and PSP operations. Then, for all VDM (meta) data structures and their respective operations, executable interpretations in terms of PSN and the PSP commands are defined in detail. Finally, it is outlined how an interpretable VDM semantic model can be constructed, and some automatic means of analyzing and testing the specifications are suggested.

4.2. INTERPRETING A VDM MODEL ON THE DMP

The process of 'translating' a formal VDM semantic model into its executable PSN/PSP representation can be divided into several steps according to the main components of the VDM specification.

4.2.1 STRUCTURE SPECIFICATION

4.2.1.1 Semantic Domains

In VDM, the object structures are defined in the 'semantic domains' part of the formal model. In the PSP, each structure is represented by a p-set, and the p-set structures are specified in detail by a PSP 'TeMPlate' command. (The upper case letters in the PSP commands refer to their respective abbreviations in the DMP.) So, for each structure definition in the VDM semantic domains, a corresponding template definition with the same name is specified in the PSP interpreter. This template definition describes all possible p-sets which can represent the elements of that respective domain. How the right hand side of a semantic domain definition translates into the p-set (template) representation, depends on its abstract structure and will be explained in detail in the following section.

If the VDM semantic domain definition makes use of the BNF-like "|" (bar) to specify different alternatives for possible structures of its elements, this definition is translated into as many alternative p-set template definitions. Thus, each actual p-set representation of any such domain element has to 'conform' to one of the corresponding templates at all times.

Then, for all abstract object types defined in the semantic domains, an (in general recursive) 'is-<object-type>' function is automatically generated by the system. It yields true for any object (i.e. p-set) instance conforming to that type, and false for all other objects. This function may be based on other 'is-<object-type>' functions, if the particular abstract object domain is specified in terms of other semantic domain objects. Finally, the 'is-<object-type>' function makes use of the 'ISIN' PSP command to test the membership of an element/duplex instance in an actual p-set which represents an abstract VDM domain.

4.2.1.2 Static Consistency Constraints

The restrictions to valid ('well formed') semantic domain objects are expressed in VDM by Boolean 'is-well-formed' functions in the static consistency constraint part of the formal model. If these 'is-well-formed' functions are expressed in a way such that each of them can be related to the specification of exactly one semantic domain object, the respective predicate is included in the 'WHERE' clause of the corresponding template definition in the PSN/PSP representation. Other static consistency predicates or auxiliary functions should not be used in this part of the VDM semantic model. (This does not restrict the expressiveness of the formal model.)

4.2.1.3 Global State

To ease the interpretation of its semantic functions, the VDM semantic model should be based on an explicit global state. Then, for each database variable which is part of the global state in the VDM semantic model, a corresponding p-set variable is declared in the PSN/PSP representation. The initial p-set value is either created by a p-set 'CReate' command, or the PSP 'POPulate' operation may initiate a dialog session to provide the initial duplex instances to the system. In either case, the interpreting system has to make sure that all values of the global variables conform at all times to the template definitions which represent their respective semantic domain object definitions.

4.2.2 OPERATION SPECIFICATIONS

4.2.2.1 VDM Model of the Operation Semantics

The abstract syntactic structure of the operations and their operands is specified (in a similar way as the object structures) in the 'syntactic domains' of the VDM semantic model. Restrictions to the operands are defined by is-well-formed predicates in the 'dynamic consistency constraints'. Each operation's semantics is then denoted by a semantic 'interpretation' function in the 'elaboration functions' specification of the formal semantic model. If necessary, there are also semantic 'evaluation' functions defined which denote the meaning of any expression evaluation.

In the PSN/PSP interpretation of the operation semantics model, the semantic functions are represented by equivalent procedures and functions. How to derive these procedures and function from the corresponding VDM semantic functions will be explained in the next two paragraphs.

4.2.2.2 Interpretation Functions

For each (operation) interpretation function of the 'elaboration functions' part of the VDM model, a procedure is defined for the PSP according to the following description:

- the <u>name</u> of the procedure is given by the name of the left-hand side of the corresponding rule in the syntactic domains of the VDM model;

- the <u>argument types</u> for the procedure are given by the p-set representations of the right-hand side of the corresponding rule in the syntactic domains of the VDM model; either these object types are already defined in the semantic domains, or they (and their p-set representations) are derived from such object types in the same way as described for the semantic domains objects in 2.1.1.

- the procedure bodies start with checking the <u>restrictions to the arguments</u> as specified in the dynamic consistency constraints of the VDM model. For each operation, the corresponding 'is-well-formed' predicate is applied to check the validity of the actual arguments of the procedure representing the operation. If this check yields _false_, the operations will not be interpreted, and some form of exception handling takes place instead.

- Otherwise, the _procedure body_ is built up by the PSP representation of the corresponding interpretation function of the VDM model. How this representation is derived from the VDM interpretation function will be described later.

The executions of the 'semantic' procedures defined so far, can (and usually do) access components of the global state. All these procedure executions result in side-effects, i.e. changes to variables of the global state. 'Auxiliary' functions may be used, whenever it is convenient, in the same way as in the VDM semantic model.

4.2.2.3 Evaluation Functions

For each evaluation function which in VDM models the semantics of evaluating expressions, a corresponding function is defined in the following way:

- its _name_ is given by the left hand side of the corresponding rule in the syntactic domains;

- its _argument (type)s_ are given by the right-hand side of the corresponding rule in the syntactic domains;

- _restrictions to the arguments_ as possibly specified _in VDM 'is-well-formed'_ functions (in the dynamic consistency constraints) are checked at the very beginning of each function evaluation;

- the _result_ of a function call is (in general) a complex value as specified by the VDM evaluation function definition, or, in case an

- _error_ occurs during the function evaluation, the _result_ is _undefined_, and some special exception handling procedures may be triggered;

- the _function body_ is made out of the (slightly restricted) VDM function specification. It derives a result value from the function input parameters and/or additional components of the global state.

No function evaluation does at any time change the global state; i.e. functions do not have side effects. All function definitions may be based on additional auxiliary functions; but note that all necessary pre-conditions for the application of these auxiliary functions have to be checked as (possibly) specified in the respective '<u>pre</u>-predicates' at the end of each corresponding VDM function specification.

4.2.2.4 Interpreting the Semantic Functions

The specification meta-language of VDM uses a set of abstract (meta) data structures to denote object structures, and semantic function specifications to express the semantics of the object systems' operations. Basically, the VDM representations for semantic functions are expressed in terms of a set of (meta) operations which are predefined for all VDM meta data structures. Some (meta) control structures may be used in a way known from usual high-level programming languages. In addition, first-order predicates add to the expressiveness of the semantic functions' specifications. Finally, some alternative (and easier to use) means of expressing certain parts of the function specifications are provided. This (meta) syntactic 'sugar', however, does not add to the specification power of the meta-language.

In order to interpret the VDM semantic function specifications on the PSP, we first have to provide interpretations for (meta) data structures and their respective operations. This will be shown in detail in the next section. The VDM (meta) control structures can be interpreted directly by the corresponding control structures of the programming language environment of the PSP (as given in the DMP and augmented by the programming language C). The use of first-order predicates has to be restricted to only those which can be represented in the 'WHERE' clauses of the PSP 'CReate' commands. The 'syntactically sugared' notations in the semantic functions' specifications can either be interpreted after transformation into their 'unsugared', basic VDM equivalents, or, as they are in most cases similar to usual programming language constructs, directly by the PSP programming language environment.

4.3. INTERPRETATIONS FOR THE META DATA STRUCTURES

4.3.1 ELEMENTARY DATA TYPES

Meta IV, the meta-language of VDM basically provides three different 'elementary' (meta) data types for the definition of semantic objects. They are interpreted by predefined sets of atomic values in PSN as follows:

- the abstract VDM data type Boolean with its corresponding value set BOOL and the set of Boolean operations will be interpreted by usual Boolean data type as available on the interpreting machine;

- the VDM data type Numeral (whose value set, NUM, comprises INTEGER and REAL values) is interpreted by the usual data types INTEGER and REAL combined;

- the VDM data type Quotation (value set QUOT) is interpreted by the (indivisible) character strings CHARACTER in PSN/PSP.

All operations (i.e. all ways of deriving expressions) in the elementary data types are interpreted by the respective usual operations of the programming language environment of the PSP.

4.3.2 COMPOSITE DATA TYPES

4.3.2.1 Sets

Any VDM abstract set, s, of elements, e_i, is interpreted by the following p-set:

- in case the set was explicitly defined by the VDM set specification, {e_1,...,e_n}, it is interpreted by the (classical) p-set {e_1@#,...,e_n@#},

- in case the set was implicitly defined by the VDM set specification, {e | e ∈ E and P(e)}, it is interpreted by the p-set-create expression
  CR WITH e.all WHERE p(e) ,
  provided a range variable, e, is declared to range over the (p-set) interpretation of the set E.

The common set operators, union, intersection, difference, cardinality, complement, membership, and subset predicate are interpreted in a straightforward way by the corresponding classical (p-) set operations, UN, IN, SY, CD, ISIN, RC, and SB. However, application of the PSP commands should be restricted to operands representing classical sets only. Whether a p-set represents a classical set can be checked by the PSP predicate CS.

All set operations are interpreted in terms of their respective PSP commands, and, in addition, possibly some high-level programming language constructs from the programming language environment of the PSP. In this section, the operation interpretations will be expressed in a way similar to usual programming language function definitions, stating first each function's name, its parameters, possibly the type of its result (only if other than p-set), and then, as the function's body, the PSP interpretation for the respective (VDM) operation. As nearly all parameters are p-sets, parameter types will be given explicitly only if other than p-set.

So, the abstract VDM set operations can be interpreted in the PSP in following way:

```
function  set_union  (pset_1,pset_2)  =
      if  CS (pset_1)  and  CS (pset_2)
          then  UN pset_1 pset_2
          else  undefined


function  set_intersection  (pset_1,pset_2)  =
      if  CS (pset_1)  and  CS (pset_2)
          then  IN pset_1 pset_2
          else  undefined


function  set_difference  (pset_1,pset_2)  =
      if  CS (pset_1)  and  CS (pset_2)
          then  SY pset_1 pset_2
          else  undefined


function  set_cardinality  (pset)  :  INTEGER  =
      if  CS (pset)
          then  CD pset
          else  undefined


function  set_element  (elem,pset)  :  BOOL  =
      if  CS (elem)  and  CS (pset)
                     and  CD elem = 1
          then  ISIN elem pset
          else  undefined


function  set_complement  (pset_1,pset_2)  =
      if  CS (pset_1)  and  CS (pset_2)
          then  RC pset_1 pset_2
          else  undefined


function  set_subset  (pset_1,pset_2)  :  BOOL  =
      if  CS (pset_1)  and  CS (pset_2)
          then  SB pset_1 pset_2
          else  undefined
```

4.3.2.2 Tuples

Abstract tuples of the VDM meta-language consisting of a list of components, c_i, are interpreted by the following p-sets:

- an explicit tuple specification <c_1,...,c_n> is represented by the p-set [[c_1@component,1@index]@#, ...], and

- for an implicit tuple specification < f(i) | $1 \leq i \leq n$ >, where f is a function mapping integers to components f: INTEGER ---> COMPONENTS, the corresponding p-set representation is created by the PSP-commands
```
    RG c IS <p-set representation of COMPONENTS>;
    RG i IS <p-set representation of INTEGERS 1...n>;
    CR WITH (component:=c, index:=i) WHERE (f(i)=c).
```

To assure the correctness of the operands for tuple operations, we have to provide ways to test whether a given p-set represents a tuple or not. Therefore, a 'tuple-template', Tpl_tdef, has to be predefined in the PSP:

```
     TMP Tpl_tdef = [[c@component, i@index]@# ...
                             WHERE (i ∈ {1,...,n})]
```

Based on this template definition, we can test whether a given p-set 'conforms' (CNF) to it or not.

Now, the selection of a single tuple component, indexed by an integer value, i, can be interpreted by

```
   function  tpl_select  (tpl, i:INTEGER)  =
         if  CNF tpl TO Tpl_tdef  and   i ≤ CD tpl
             then { RG d IS tpl;
                       CR WITH (d.component) WHERE (d.index=i) }
             else  undefined
```

The other VDM tuple operations, len (number of components), hd (first component), tl (tuple without first component), elems (set of components), ind (index set), update (change of a component as given by its index), and concatenate (concatenates two tuples), are interpreted as follows:

```
   function  tpl_len  (tpl)   =
         if  CNF tpl TO Tpl_tdef
             then  CD tpl
             else  undefined
```

```
function  tpl_hd  (tpl)  =
      if  CNF tpl TO Tpl_tdef
          then  tpl_select (tpl,1)
          else  undefined


function  tpl_tl  (tpl)  =
      if  CNF tpl TO Tpl_tdef
          then { RG d IS tpl;
                 CR WITH (d.component,index:=d.index-1)
                    WHERE (d.index ≠ 1) }
          else  undefined


function  tpl_elems  (tpl)  =
      if  CNF tpl TO Tpl_tdef
          then { RG d IS tpl;
                 RG c IS d.component;
                 CR WITH (c.all) }
          else  undefined


function  tpl_ind  (tpl)  =
      if  CNF tpl TO Tpl_tdef
          then { RG d IS tpl;
                 RG i IS d.index;
                 CR WITH (i.all) }
          else  undefined


function  tpl_update  (tpl,i:integer,new)  =
      if  CNF tpl TO Tpl_tdef  and  CNF new TO Tpl_tdef
          and  i < CD tpl_elems (tpl)
          and  tpl_ind (new) = [i@#]
          then { RG e IS tpl;
                 CR temp WITH (e.component)
                    WHERE (e.index ≠ i);
                 UN temp new }
          else   undefined


function  tpl_concatenate  (tpl_1,tpl_2)  =
      if  CNF tpl_1 TO Tpl_tdef  and
          CNF tpl_2 TO Tpl_tdef
          then { RG e IS tpl_2;
                 CR temp WITH (e.component,index:=
                                  e.index + tpl_len (tpl_1));
                 UN tpl_1 temp }
          else  undefined
```

4.3.2.3 Maps

A VDM map, m, mapping domain elements, a_i ∈ A, to corresponding range elements, b_i ∈ B, is represented in the PSP by one of these methods:

- in case it is specified explicitly as m = [a_1 --> b_1, ..., a_n --> b_n] by the p-set [[a_1@argument, b_1@result]@#...], or

- in case it is specified implicitly in terms of a predicate, p: (A,B)--->BOOL, m=[a-->b | p(a,b)], its p-set representation, m', is created by the following PSP commands:

```
          RG a IS A;  RG b IS B;
          CR m' WITH (argument:=a,result:=b)
                WHERE ( p(a,b) )
```

For type checking purposes, again, a 'map_template' is defined in the PSP as

```
     TMP Map_tdef = [[a@argument,b@result]@#...]
         WHERE CD Map_tdef = CD ( CR WITH (mtd.argument) )
```

provided 'RG mtd IS Map_tdef'.

The VDM operations defined for maps, rng (range), dom (domain), apply (application of a map to an argument), restrict (restriction of the domain of a map to a subset), exclude (exclusion of a subset of a map's domain), alter (extension and updating of a map by another map), compose (functional composition of two maps), and union (union of two maps with disjunct domains) are interpreted by the following functions:

```
function  map_dom  (map)  =
      if  CNF map TO Map_tdef
          then { RG p IS map;
                 CR WITH (p.argument) }
          else   undefined


function  map_rng  (map)  =
      if  CNF map TO Map_tdef
          then { RG p IS map;
                 CR WITH (p.result) }
          else   undefined
```

```
function  map_apply  (map,arg)  =
      if  CNF map TO Map_tdef  and
          CS (arg)  and  CD (arg) = 1
          then { RG p IS map;
                 CR WITH (p.result)
                    WHERE (p.argumen =arg) }
          else   undefined


function  map_restrict  (map,set)  =
      if  CNF map TO Map_tdef  and  CS (set)
          then { RG p IS map;
                 CR WITH (p.all) WHERE
                  ( ISIN  p.argument  map_dom (map) ) }
          else   undefined


function  map_exclude  (map,set)  =
      if  CNF map TO Map_tdef  and  CS (set)
          then { RG p IS map;
                 CR WITH (p.all) WHERE NOT
                  ( ISIN  p.argument  map_dom (map) ) }
          else   undefined


function  map_alter  (map_1,map_2)  =
      if  CNF map_1 TO Map_tdef  and
          CNF map_2 TO Map_tdef
          then { RG p IS map_1;
                 CR temp WITH (p.all) WHERE NOT
                  ( ISIN  p.argument  map_dom (map_2) );
                 UN temp map_2 }
          else   undefined


function  map_compose  (map_1,map_2)  =
      if  CNF map_1 TO Map_tdef  and
          CNF map_2 TO Map_tdef  and
          SB map-rng (map_1) map_dom (map_2)
          then { RG p_1 IS map_1; RG p_2 IS map_2;
                 CR WITH ( p_1.argument, p_2.result )
                    WHERE ( p_1.result = p_2.argument ) }
          else   undefined


function  map_union  (map_1,map_2)  =
      if  CNF map_1 TO Map_tdef  and
          CNF map_2 TO Map_tdef  and
          NN  ( IN map_dom (map_1)  map_dom (map_2) )
          then  UN map_1 map_2
          else   undefined
```

4.3.2.4 Trees

An abstract VDM tree, t=(d_1,...,d_n), from the domain D :: D_1...D_n is represented by the p-set [[d_1@branch, D_1@domain]@#...].

Again, for type checking purposes, a tree template is defined as

```
   TMP Tree_tdef = [[d@branch,D@domain]@#...WHERE ISIN d D].
```

There are only two different operations to be described for trees: the construction of a new tree from the domain D, mk-D (d_1,...,d_n), and the selection of a component, s-D_i.

```
   function  tree_selection  (tree,domain) =
         if  CNF tree TO Tree_tdef  and
             CS (domain)  and  CD (domain) = 1
             then { RG b IS tree;  RG d IS domain;
                    CR WITH ( b.branch )
                       WHERE ( b.domain = d ) }
             else   undefined

   function  tree_make  (d_list,D_list) =
         if  CNF d_list TO Tpl_tdef  and
             CNF D_list TO Tpl_tdef  and
             tpl-len (d_list) = tpl_len (D_list)
             then { RG d IS d_list;  RG D IS D_list;
                    CR WITH (branch:=d.component, domain:=
                       D.component) WHERE (d.index = D.index) }
             else   undefined
```

4.3.2.5 Functions

The VDM sets which can be represented on any limited, real-world computer always have to be finite. In the same way, all specified domains have to be limited as well, even if they are defined recursively (i.e. an upper bound can be placed on the depth of any recursion). This, however, means that all function domains are also finite. Therefore we can represent all abstract VDM functions by abstract maps. There is, thus, no explicit need for an extra function interpretation besides the presented interpretation for maps in the VDM interpreter. In order to make the interpretations simpler, we will, however, represent all semantic functions by procedures and (PSP) functions as described in subsection 4.2.2. The only function operation, the application to its arguments, is represented by the function or procedure execution in the VDM interpreter.

4.4. USE OF THE INTERPRETABLE SPECIFICATIONS

4.4.1 SPECIFYING AN OBJECT SYSTEM'S SEMANTICS

The interpretable specifications as described in the preceding sections can be used in a similar way as the data model specifications in the data model processor [4,5] to define an object system's semantics formally. The meta-language to be used, however, will be VDM's Meta IV rather than PSN and the original PSP commands.

First, all semantic object types are defined in the semantic domains specifications of the formal model. Then, each semantic domain definition may be refined by an additional is-well-formed predicate in the static consistency constraints section of the semantic model. Auxiliary functions may be defined for and applied in any of the predicate definitions.

Next, the global state has to be defined. In this part of the formal semantic model, a global state variable has to be declared for each object which might be affected during the interpretation of any of the operations to be specified. In addition, global state constants might be defined for all objects which are accessible from at least one operation, but which will not be altered by any of them. The types of all global variables and constants have, in any case, to be defined in, or to be derived from the semantic domains.

Initializing the global state variables and constants is a task comparable to the 'data model populator' role in the DMP. An initial value has to be assigned to all variables and constants. All the values have to conform to both the corresponding structure descriptions in the semantic domains, and, possibly, to their static consistency constraints.

In the second major part of the formal semantic model, first the operations' names and operand types have to be declared in the syntactic domains part of the model. All operand types are either defined in the semantic domains, or their definitions have to be given here. (The most important example for such operand type definitions are the expression derivation rules to be specified in the syntactic domains.)

Then, for each operation a dynamic consistency predicate may be defined. It restricts the set of all possible operands (as specified in the syntactic domains) to the only well formed ones for, possibly, each single operation.

Finally, the semantic functions are defined which describe the semantics for all operations. For each operation to be specified the corresponding semantic interpretation function takes this operation's arguments, and defines its semantics in terms of 'manipulating' side effects to some global state variables. The semantic functions which denote expression evaluations do not manipulate any global state variables. They describe (in an abstract way) the derivation of a result value from the global variables and/or constants. Auxiliary functions may be used in any semantic function definition.

4.4.2 ANALYZING THE INTERPRETABLE SPECIFICATIONS

Once a given object system is defined formally in terms of the (slightly restricted) VDM meta-language, we are now able to transform the semantic model into a semantically equivalent PSN/PSP representation as described in sections 4.2 and 4.3. Then, the formal model can be analyzed automatically on the DMP.

First, some consistency and completeness checks could be applied:

- In the semantic domains, all object names (left-hand sides of the defining equations) have to be defined uniquely. For all object names used in the defining expressions (right-hand sides of the defining equations), a defining rule has to exist in the semantic domains.

- Similarly, in the syntactic domains all operation names have to be defined uniquely. Also, each defining expression may be based on already defined semantic domain objects only or on other abstract syntactic objects specified in the syntactic domains.

- There may be one Boolean 'is-well-formed' function specified for each semantic or syntactic domain object (i.e. for each semantic object or operation). All these predicates can possibly be checked for consistency, i.e. not to be so 'restrictive' that no single semantic or syntactic domain object can make them <u>true</u>.

- Most important, it should be checked that each actual variable or constant in the global state conforms to all structural restrictions as imposed by the corresponding semantic domain definitions. In addition, the corresponding static consistency predicate should yield true if applied to each of them. The consistency checks should be repeated for each global variable after any manipulation during a semantic function interpretation. In case a global value is 'incorrect' by that means, appropriate exception handling procedures should be triggered (at least a message to the user of the interpreting system).

- For each operation specified in the syntactic domains, there should be exactly one interpretation function in the elaboration functions' part of the formal model. Each of these functions should only take parameters of the correct types as specified in the syntactic domains. Interpretation functions should have no result values but rather have side-effects on some global state variables. For each possible kind of expression there should be exactly one semantic evaluation function which, again, takes the correct arguments, does not change any global state variable, and results in a value of the appropriate type.

All mentioned restrictions could be checked automatically by the formal semantics interpreting system.

Finally, a formal specification of an object system's semantics could be verified against its intended, and usually informal initial semantic description (even if this is not much more than some rather 'vague' ideas about the system). Any specified operation can be simulated based on its respective VDM semantic function specification. First, the correctness of its parameters is assured, and then the function is interpreted on the PSP. The semantic meaning of the operation is defined by the global state changes resulting from the interpretation of the corresponding semantic function. If these changes do not reflect the initial understanding of the operation's semantics, the specification has to be changed and interpreted again in an iterative process.

5. CONCLUSIONS

This paper introduces two different approaches to database model semantics specification. In chapter 2, both of them are applied to a subset of the Relational Data Model. A comparison of their main advantages and disadvantages leads to two alternative approaches to integration of both methods.

In chapter 3, the Vienna Development Method is applied to specify the semantics of the Positional Set Notation and the corresponding Positional Set Processor commands of the Data Model Processor. It is shown that Meta IV, the meta-language of VDM is powerful enough to express all semantic properties of the most important constructs of the data model interpreter's specification language in a completely formal way.

VDM is demonstrated to be an appropriate tool to model all three semantic aspects of PSN and the PSP commands: structures, operations, and both kinds of possible constraints. The result is a formal model which is fairly simple and straightforward for both the structure and constraint descriptions. The functional model for the operations is slightly more complex, especially for some of the PSP commands. (See, for instance, the abstract syntax and elaboration function for the 'create' command.) This may be interpreted as a measure for complexity of the operations which may, thus, not be so easily understood by novice users of the DMP. However, breaking the complex semantic functions down into several simpler (auxiliary) ones should help to guide the user's understanding.

For any semantic specification method, the semantic definition of its meta-language tools is its crucial basis. In the DMP framework, the meta-language tools are given by PSN and the PSP commands. So, the precise and formal VDM semantic model provides the semantic basis for the data model specification language of the DMP using a theoretically sound and widely accepted approach to formal semantic specification.

In chapter 4, it is shown how a formal semantic specification can be written in an abstract and high-level meta-language, and still be analyzed and evaluated immediately on a software system. This approach combines the major advantages of both VDM and the PSP into a fairly understandable and powerful prototyping system.

The scope of the abstract semantic specification interpreter is centered around but not limited to database models. Data model semantics can be defined in an abstract and representation-free meta-language, and then be emulated on the data model processor. By using the integrated methods, single database management systems' user interfaces, or particular database applications with corresponding transaction specifications can also be defined at a high level of abstraction, and then be tested and analyzed automatically at a rather early stage of their design.

ACKNOWLEDGEMENTS

The help from Drs. D.R. Deutsch, L. Gallagher, W.T. Hardgrave, M. Koll, J.W. Schmidt, and Ms. S.B. Salazar in supporting and reviewing this work is gratefully acknowledged.

6. REFERENCES

[1] D. Bjørner, C.B. Jones (Eds.): 'The Vienna Development Method: The Meta Language', Lecture Notes in Computer Science, no. 61, Springer Verlag, Berlin Heidelberg New York, 1978.

[2] M.L. Brodie, J.W. Schmidt (Eds.): 'Final Report of the Relational Database Task Group', ANSI/X3/SPARC/DBSSG, ACM SIGMOD RECORD, vol. 12, no. 4, July 1982.

[3] W.T. Hardgrave: 'Positional Set Notation', Advances in Database Management, vol. 2, Heyden and Son, New York, to appear 1983.

[4] W.T. Hardgrave, S.B. Salazar : 'The Positional Set Processor: A Tool for Data Modeling', NBSIR 81-2302, National Bureau of Standards, Washington, D.C., 1981.

[5] M. Koll, W.T. Hardgrave, S.B. Salazar: 'Data Model Processing', Proc. National Computer Conference, Houston, Texas, June 1982.

[6] M. Koll, W.T. Hardgrave: 'Data Model Processor User's Manual', Institute for Computer Science and Technology, Center for Programming Science and Technology, Data Management and Programming Language Division, National Bureau of Standards, Washington, D.C., September 1981.

[7] W. Lamersdorf, J.W. Schmidt: 'Specification of Pascal/R', Berichte Nr. 73 and 74, Fachbereich Informatik, Universität Hamburg, West-Germany, 1980.

[8] E.J. Neuhold, T. Olnhoff : 'The Vienna Development Method (VDM) and its Use for the Specification of a Relational Data Base System', T. Lavington (Ed.) : Proc. IFIP 80, World Computer Congress, Tokyo/Melbourne, Oct. 1980, pp.3-16.

[9] J.W. Schmidt, W. Lamersdorf: 'Relational Datamodel: A Definition and its Formalization', Bericht Nr. 88, Fachbereich Informatik, Universität Hamburg, West Germany, March 1982 - also available in [2].

NBS-114A (REV. 2-80)

U.S. DEPT. OF COMM. BIBLIOGRAPHIC DATA SHEET *(See instructions)*	1. PUBLICATION OR REPORT NO. NBSIR 83-2740	2. Performing Organ. Report No. Hamburg-University DBPL-Memo 111-82	3. Publication Date July 1983

4. TITLE AND SUBTITLE

Specification and Interpretation of Data Model Semantics: An Integration of Two Approaches

5. AUTHOR(S)

Winfried Lamersdorf

6. PERFORMING ORGANIZATION *(If joint or other than NBS, see instructions)*

NATIONAL BUREAU OF STANDARDS
DEPARTMENT OF COMMERCE
WASHINGTON, D.C. 20234

7. Contract/Grant No.

8. Type of Report & Period Covered

9. SPONSORING ORGANIZATION NAME AND COMPLETE ADDRESS *(Street, City, State, ZIP)*

10. SUPPLEMENTARY NOTES

[ ] Document describes a computer program; SF-185, FIPS Software Summary, is attached.

11. ABSTRACT *(A 200-word or less factual summary of most significant information. If document includes a significant bibliography or literature survey, mention it here)*

Two different approaches to database model description and evaluation are presented, compared, and integrated: a formal semantic specification method as originally developed for programming languages, and a computer-based data model processor to be used as a rapid prototyping system. Both ways to specify database model semantics are applied to a common example.

Two alternatives to combine the advantages of both methods are analyzed in detail. First, it is shown how the semantics of the data model processor can be precisely described in terms of the formal semantic specification method. Then, it is demonstrated how the abstract meta-language of the specification method can be mapped to the executable commands of the data model processor. Thus, database semantics can both be specified in an abstract and high-level way and still be analyzed and evaluated automatically.

12. KEY WORDS *(Six to twelve entries; alphabetical order; capitalize only proper names; and separate key words by semicolons)*

databases; data models; data model processing; data model prototyping; data model semantics; denotational semantics; formal semantic specification; relational database; relational data model; semantic model interpreter

13. AVAILABILITY

[X] Unlimited
[ ] For Official Distribution. Do Not Release to NTIS
[ ] Order From Superintendent of Documents, U.S. Government Printing Office, Washington, D.C. 20402.
XX Order From National Technical Information Service (NTIS), Springfield, VA. 22161

14. NO. OF PRINTED PAGES

56

15. Price

$10.00